



SymmetricDS User Guide

User Guide 1.0

Copyright © 2007, 2008 Eric Long, Chris Henson

Permission to use, copy, modify, and distribute the SymmetricDS User Guide Version 1.0 for any purpose and without fee is hereby granted in perpetuity, provided that the above copyright notice and this paragraph appear in all copies.

Table of Contents

Preface	v
1. Introduction	1
1.1. Introduction	1
1.1.1. What is SymmetricDS?	1
1.1.2. Notification Schemes	1
1.1.3. Two-Way Table Synchronization	1
1.1.4. Data Channels	1
1.1.5. Transaction Awareness	2
1.1.6. Data Filtering and Rerouting	2
1.1.7. HTTP Transport	2
1.1.8. Remote Management	3
1.2. Requirements	3
1.3. Background	3
2. Getting Started	5
2.1. Installing SymmetricDS	5
2.2. Creating and Populating Your Databases	6
2.3. Starting SymmetricDS	7
2.4. Registering a Node	8
2.5. Sending Initial Load	8
2.6. Pulling Data	8
2.7. Pushing Data	9
2.8. Verifying Outgoing Batches	9
2.9. Verifying Incoming Batches	10
3. Concepts	12
3.1. Configuration Data Model	12
3.1.1. Node Table	12
3.1.2. Node Security Table	13
3.1.3. Node Identity Table	13
3.1.4. Node Group Table	14
3.1.5. Node Group Link Table	14
3.1.6. Channel Table	14
3.1.7. Node Channel Control Table	15
3.1.8. Trigger Table	15
3.1.9. Parameter Table	17
3.1.10. Registration Redirect Table	17
3.2. Runtime Data Model	18
3.2.1. Trigger History Table	18
3.2.2. Data Table	19
3.2.3. Data Event Table	19
3.2.4. Outgoing Batch Table	20
3.2.5. Outgoing Batch History Table	20
3.2.6. Incoming Batch Table	21
3.2.7. Incoming Batch History Table	22
4. Architecture	23

4.1. Deployment Options	24
4.1.1. Web Archive	25
4.1.2. Standalone	26
4.1.3. Embedded	27
5. Basic Configuration	28
5.1. Basic Properties	28
5.2. Node Group	29
5.3. Node Group Link	29
5.4. Node	30
5.5. Channel	30
5.6. Trigger	31
5.7. Using Sync Conditions	31
5.8. Using Node Select	32
5.9. Using Initial Load Select	33
6. Advanced Configuration	34
6.1. Multi-Tiered Synchronization	34
6.2. Bi-Directional Synchronization	34
6.3. Dead Triggers	35
6.4. Database Purging	35
6.5. Extension Points	35
6.5.1. IParameterFilter	36
6.5.2. IDataLoaderFilter	36
6.5.3. ITableColumnFilter	38
6.5.4. IBatchListener	38
6.5.5. IAcknowledgeEventListener	38
6.5.6. IReloadListener	38
6.5.7. IExtractorFilter	38
6.5.8. ISyncUrlExtension	38
6.5.9. INodeIdGenerator	38
6.5.10. ITriggerCreationListener	38
6.6. Encrypted Passwords	39
6.7. Secure Transport	39
6.7.1. Sym Launcher	39
6.7.2. Tomcat	39
6.7.3. Keystores	40
6.7.4. Generating Keys	40
7. Administration	42
7.1. Changing Triggers	42
7.2. Changing Configuration	42
7.3. Sync Triggers Job	42
7.4. Enabling and Disabling Synchronization	43
7.5. Java Management Extensions	43
7.6. Opening Registration	44
7.7. Viewing Batches	44
7.8. Statistics	44
7.9. Clustering	44
7.10. Running as a Windows Service	45
7.11. Running as a Unix Service	45
A. Parameters	47

A.1. Startup Parameters	47
A.2. Runtime Parameters	49
B. Database Notes	53
B.1. Oracle	53
B.2. MySQL	54
B.3. PostgreSQL	54
B.4. MS SQL Server	55
B.5. HSQLDB	55
B.6. H2	55
B.7. Apache Derby	55
B.8. IBM DB2	55
B.9. Firebird	56
C. Data Format	57
D. Version Numbering	59

Preface

SymmetricDS is web-enabled, database independent, data synchronization software. It uses web and database technologies to replicate tables between relational databases in near real time. The software was designed to scale for a large number of databases, work across low-bandwidth connections, and withstand periods of network outage.

This User Guide describes the SymmetricDS library for data synchronization. It is intended for users who want to be quickly familiarized with the software, configure it, and use its features.

Chapter 1. Introduction

1.1. Introduction

1.1.1. What is SymmetricDS?

SymmetricDS is asynchronous data replication software supporting multiple subscribers and bi-directional synchronization. It uses web and database technologies to replicate tables between relational databases in near real time. The software was designed to scale for a large number of databases, work across low-bandwidth connections, and withstand periods of network outage.

The software is installed either standalone, as a web application in a Java application server or can be embedded into another Java application. A single installation of SymmetricDS is called a *Node*. A Node is initialized by a properties file and is configured by inserting configuration data into a series of database tables. It then creates database triggers on the application tables specified so that database events are captured for delivery to other SymmetricDS nodes.

SymmetricDS supports synchronization across different database platforms through the concept of database dialects. A database dialect is an abstraction layer that SymmetricDS interacts with to insulated the main synchronization logic from database specific implementation details.

1.1.2. Notification Schemes

After a change to the database is recorded, the nodes interested in the change are notified. Change notification is configured to perform a push (trickle-back) or a pull (trickle-poll) of data. When several nodes target their changes to a central node, it is efficient to push the changes instead of waiting for the central node to pull from each source node. When network configuration protects a node with a firewall, a pull configuration allows the node to receive data changes that might otherwise be blocked using push. The frequency of the change notification is configured by default to one minute.

1.1.3. Two-Way Table Synchronization

Some data may synchronize in one direction. For example, a retail store sends its sales transactions to a central office, and the central office sends its stock items to the store. Some data may synchronize in both directions. For example, the retail store sends the central office an inventory document, and the central office updates the document status, which is sent back to the store. SymmetricDS supports bi-directional synchronization and avoids getting into update loops by only recording data changes outside of synchronization.

1.1.4. Data Channels

Data synchronization is defined at the table (or table subset) level. Each managed table can be assigned to a channel that helps control the flow of data. A channel is a category of data that can be enabled,

prioritized and synchronized independently of other channels. For example, in a retail environment, users may be waiting for inventory documents to update while a promotional sale event updates a large number of items. If processed in order, the item updates would delay the inventory updates even though the data is unrelated. By assigning item table changes to the "item" channel and inventory table changes to the "inventory" channel, the changes are processed independently so inventory can get through.

1.1.5. Transaction Awareness

Many databases provide a unique transaction identifier associated with the rows that are committed together. SymmetricDS stores the transaction ID along with the data that changed so it can play back the transaction exactly the way it happened. This means the target database maintains the same integrity as its source. Support for transaction ID is documented in the appendix of this guide.

1.1.6. Data Filtering and Rerouting

Data can be filtered as it is recorded, extracted, and loaded.

- Data routing and filtering is specified using SQL expressions in the trigger configuration. The SQL expressions are configured in the `node_select` (for real time routing) and `initial_load_select` (for initial data loads) columns of the Section 3.1.8, "Trigger Table". The SQL expressions are used to create the SQL triggers that SymmetricDS installs to capture data changes. Using this technique data can be routed to a specific node or group of nodes.
- As data changes are loaded in the target database, a class implementing `IDataLoaderFilter` can change the data in a column or route it somewhere else. One possible use might be to route credit card data to a secure database and blank it out as it loads into a centralized sales database. The filter can also prevent data from reaching the database altogether, effectively replacing the default data loading.
- Columns can be excluded from synchronization so they are never recorded when the table is changed. As data changes are loaded into the target database, a class implementing `IColumnFilter` can altogether remove a column from the synchronization. For example, an employee table may be synchronized to a retail store database, but the employee's password is only synchronized on the initial insert.
- As data changes are extracted from the source database, a class implementing the `IExtractorListener` interface is called to filter data or route it somewhere else. By default, SymmetricDS provides a handler that transforms and streams data as CSV. Optionally, an alternate implementation may be provided to take some other action on the extracted data.

1.1.7. HTTP Transport

By default, SymmetricDS uses web-based HTTP in a style called Representation State Transfer (REST) that is lightweight and easy to manage. A series of filters are also provided to enforce authentication and to restrict the number of simultaneous synchronization streams. The `ITransportManager` interface allows other transports to be implemented. (The unit tests for SymmetricDS take advantage of this by using an

`InternalTransportManager` that makes it easy to run automated tests locally.)

1.1.8. Remote Management

Administration functions are exposed through Java Management Extensions (JMX) that can be accessed from the Java JConsole or through an application server. Functions include opening registration, reloading data, purging old data, and viewing batches. A number of configuration and runtime properties are available to be viewed as well.

SymmetricDS also provides functionality to send a SQL events through the same synchronization mechanism that is used to send data. The data payload can be any SQL statement. The event is processed and acknowledged just like any other event type.

1.2. Requirements

SymmetricDS is written in Java 5 and requires a Java SE Runtime Environment (JRE) or Java SE Development Kit (JDK) version 5.0 or above.

Any database with trigger technology and a JDBC driver has the potential to run SymmetricDS. The database is abstracted through a Database Dialect in order to support specific features of each database. The following Database Dialects have been included with this release:

- MySQL version 5.0.2 and above
- Oracle version 8.1.7 and above
- PostgreSQL version 8.2.5 and above
- Sql Server 2005
- HSQLDB 1.8
- H2 1.1
- Apache Derby 10.3.2.1 and above
- IBM DB2 9.5
- Firebird 2.0 and above

See the appendix Database Notes for compatibility notes and other details for your specific database.

1.3. Background

While implementing a commercial Point of Sale (POS) system for a large retailer, the development team concluded that the software available for trickling back transactions to the general office did not meet the project needs. The list of problems in the requirements made finding the ideal solution difficult:

- Sending and receiving data with 2000 stores during peak holiday loads.
- Supporting one database platform at the store and another at general office.
- Synchronizing some data in one direction, and other data in both directions.
- Filtering out sensitive data and re-routing it to a protected database.
- Preparing the store database with an initial load of data from general office.

The team created a custom solution that met the requirements and made the project successful. From this initial challenge came the knowledge and experience that SymmetricDS benefits from today.

Chapter 2. Getting Started

This chapter is a hands-on tutorial that demonstrates how to synchronize the sample database between two running instances of SymmetricDS. This example models a retail business that has a central office database (called "root") and multiple retail store databases (called "client"). The root database sends changes to the client for item data, such as item number, description, and price. The client database sends changes to the root for sale transaction data, such as time of sale and items sold. The sample configuration specifies synchronization with a pull method for the client to receive data from root, and a push method for the root to receive data from client.

To get started, we create separate databases for the root and client where sample tables will be created and populated. We use a configuration file to run an instance of SymmetricDS, called a node, on each database. To link the nodes together, we register the client node with the root node. For this tutorial, the root database is pre-populated with data, while the client database is left empty. To load the data on the client database, we request the root node sends a "reload" to the client node. With the two databases in sync, we make changes to data in the tables and observe the changes being synchronized.

2.1. Installing SymmetricDS

Install the SymmetricDS software and configure it with your database connection information.

1. Download the [symmetric-ds-1.6.X.zip](http://www.symmetricds.org/symmetric-ds-1.6.X.zip) file from <http://www.symmetricds.org/>
2. Unzip the file in any directory you choose. This will create a `symmetric-ds-1.6.X` subdirectory, which corresponds to the version you downloaded.
3. Edit the database properties in the following property files for the root and client nodes:
 - `samples/root.properties`
 - `samples/client.properties`
4. Set the following properties in *both* files to specify how to connect to the database:

```
# The class name for the JDBC Driver
db.driver=com.mysql.jdbc.Driver

# The JDBC URL used to connect to the database
db.url=jdbc:mysql://localhost/sample

# The user to login as who can create and update tables
db.user=symmetric

# The password for the user to login as
db.password=secret
```

5. Set the following property in the `client.properties` file to specify where the root node can be contacted:

```
# The HTTP URL of the root node to contact for registration
```

```
registration.url=http://localhost:8080/sync
```

For the tutorial, the client database starts out empty, and the node is not registered. Registration is the process where the node receives its configuration and stores it in the database. The configuration describes which database tables to synchronize and with which nodes. When an unregistered node starts up, it will register with the node specified by the registration URL. The registration node centrally controls nodes on the network by allowing registration and returning configuration. In this tutorial, the registration node is the root node, which also participates in synchronization with other nodes.

2.2. Creating and Populating Your Databases



Important

You must first create the databases for your root and client nodes using the administration tools provided by your database vendor. Make sure the name of the databases you create match the settings in the properties files.

See the appendix Database Notes for compatibility with your specific database.

Create the sample tables in the *root* node database, load the sample data, and load the sample configuration.

1. Open a command prompt and navigate to the `samples` subdirectory of your SymmetricDS installation.
2. Create the sample tables in the root database.

```
../bin/sym -p root.properties --run-ddl create_sample.xml
```

Note that the warning messages from the command are safe to ignore.

3. Create the SymmetricDS tables in the root node database. These tables will contain the configuration for synchronization. The following command uses the auto-creation feature to create all the necessary SymmetricDS system tables.

```
../bin/sym -p root.properties --auto-create
```

4. Load the sample data and configuration into the root node database.

```
../bin/sym -p root.properties --run-sql insert_sample.sql
```

Create the sample tables in the *client* node database to prepare it for receiving data.

1. Open a command prompt and navigate to the `samples` subdirectory of your SymmetricDS installation.
2. Create the sample tables in the client database.

```
../bin/sym -p client.properties --run-ddl create_sample.xml
```

Note that the warning messages from the command are safe to ignore.

Verify *both* databases by logging in and listing the tables.

1. Find the item tables that sync from root to client: `item` and `item_selling_price`.
2. Find the sales tables that sync from client to root: `sale_transaction` and `sale_return_line_item`.
3. Find the SymmetricDS system tables, which have a prefix of `"sym_"`.

2.3. Starting SymmetricDS

Start the SymmetricDS nodes and observe the logging output.

1. Open a command prompt and navigate to the `samples` subdirectory of your SymmetricDS installation.
2. Start the root node server.

```
../bin/sym -p root.properties --port 8080 --server
```

The root node server starts up and creates all the triggers that were configured by the sample configuration. It listens on port 8080 for synchronization and registration requests.

3. Start the client node server.

```
../bin/sym -p client.properties --port 9090 --server
```

The client node server starts up and uses the auto-creation feature to create the SymmetricDS system tables. It begins polling the root node in order to register. Since registration is not yet open, the client node receives an authorization failure (HTTP response of 403).



Tip

If you want to change the port number used by SymmetricDS, you need to also set the `my.url` runtime property to match. The default value is:

```
my.url=http://localhost:8080/sync
```

2.4. Registering a Node

Open registration for the client node using the root node administration feature.

1. Open a command prompt and navigate to the `samples` subdirectory of your SymmetricDS installation.
2. Open registration for the client node server.

```
../bin/sym -p root.properties --open-registration "store,1"
```

The registration is opened for a node group called "store" with an external identifier of "1". This information matches the settings in `client.properties` for the client node. Each node is assigned to a group and is given an external ID that makes sense for the application. In this tutorial, we have retail stores that run SymmetricDS, so we named our group "store" and we used numeric identifiers starting with "1".

3. Watch the logging output of the client node to see it successfully register with the root node. The client is configured to attempt registration each minute. Once registered, the root and client are enabled for synchronization.

2.5. Sending Initial Load

Send an initial load of data to the client node using the root node administration feature.

1. Open a command prompt and navigate to the `samples` subdirectory of your SymmetricDS installation.
2. Send an initial load of data to the client node server.

```
../bin/sym -p root.properties --reload-node 1
```

With this command, the root node queues up an initial load for the client node that will be sent the next time the client performs its pull. The initial load includes data for each table that is configured for synchronization.

3. Watch the logging output of both nodes to see the data transfer. The client is configured to pull data from the root each minute.

2.6. Pulling Data

Modify data in the root database. The changes are propagated to the client database during pull

synchronization.

1. Open an interactive SQL session with the root database.
2. Add a new item for sale:

```
insert into item_selling_price (price_id, price) values (55, 0.65);
```

```
insert into item (item_id, price_id, name) values (110000055, 55, 'Soft Drink');
```

Once the statements are committed, the data change is captured and queued for the client node to pull.

3. Watch the logging output of both nodes to see the data transfer. The client is configured to pull data from the root each minute.
4. Verify that the new data arrives in the client database using another interactive SQL session.

2.7. Pushing Data

Modify data in the client database. The changes are propagated to the root database during push synchronization.

1. Open an interactive SQL session with the client database.
2. Add a new sale to the client database:

```
insert into sale_transaction (tran_id, store, workstation, day, seq) values (1000, '1', '3', '2007-11-01', 100);
```

```
insert into sale_return_line_item (tran_id, item_id, price, quantity) values (1000, 110000055, 0.65, 1);
```

Once the statements are committed, the data change is captured and queued for the client node to push.

3. Watch the logging output of both nodes to see the data transfer. The client is configured to push data to the root each minute.

2.8. Verifying Outgoing Batches

A batch is used for tracking and sending data changes to nodes. The sending node creates a batch and the receiving node acknowledges it. A batch in error is retried during synchronization attempts, but only after data changes in other channels are allowed to be sent. Channels are categories assigned to tables for the purpose of independent synchronization and control. Batches for a channel are not created when a batch in the channel is in error status.

1. Open an interactive SQL session with either the root or client database.
2. Verify that the data change was captured:

```
select * from sym_data where table_name like 'item%' or table_name like 'sale%';
```

Each row represents a row of data that was changed. The event_type is "I" for insert, "U" for update, or "D" for delete. For insert and update, the captured data values are listed in row_data. For update and delete, the primary key values are listed in pk_data.

3. Verify that the data change was routed to a node, using the data_id from the previous step:

```
select * from sym_data_event where data_id = ?;
```

When the batched flag is set, the data change is assigned to a batch using a batch_id that is used to track and synchronize the data. Batches are created and assigned during a push or pull synchronization.

4. Verify that the data change was batched, sent, and acknowledged, using the batch_id from the previous step:

```
select * from sym_outgoing_batch where batch_id = ?;
```

A batch represents a collection of changes to be sent to a node. The batch is created during a push or pull synchronization, when the status is set to "NE" for new. The receiving node acknowledges the batch with a status of "OK" for success or "ER" for error.

5. Verify that the batch history was recorded, using the batch_id from the previous step:

```
select * from sym_outgoing_batch_hist where batch_id = ?;
```

Work performed on the batch is recorded in the history table. A new batch with status of "NE" records the number of data changes it contains in the data_event_count field. The status of "SE" shows that a batch is being sent to a node. The acknowledgement status from the receiving node is also recorded. If the status is error, the failed_data_id indicates which row in sym_data caused the error.

2.9. Verifying Incoming Batches

The receiving node keeps track of the batches it acknowledges and records statistics about loading the data. Duplicate batches are skipped by default, but this behavior can be changed with the incoming.batches.skip.duplicates runtime property.

1. Open an interactive SQL session with either the root or client database.
2. Verify that the batch was acknowledged, using a batch_id from the previous section:

```
select * from sym_incoming_batch where batch_id = ?;
```

A batch represents a collection of changes loaded by the node. The sending node that created the batch is recorded. The status is either "OK" for success or "ER" for error.

3. Verify that the batch history was recorded, using the batch_id from the previous step:

```
select * from sym_incoming_batch_hist where batch_id = ?;
```

Work performed on the batch is recorded in the history table. If a duplicate batch was skipped, the status is recorded as "SK". Otherwise, the status is either "OK" for success or "ER" for error. The statement_count is the number of rows loaded and the byte_count is the size of the batch in bytes. The database_millis is the amount of time in milliseconds spent loading data in the database.

Chapter 3. Concepts

Many of the data synchronization concepts can be understood by examining the data model. The configuration data model is stored in a set of tables updated by the user as needed to configure the system. In contrast, the set of tables for the runtime data model change constantly as the system captures data changes and records the activity to deliver them.

Note that all tables are prepended with a configurable prefix so that multiple instances of SymmetricDS may coexist in the same database. The default prefix is *sym_*.

3.1. Configuration Data Model

The configuration is entered by the user into the data model to control the behavior of what data is synchronized to which nodes. To ease management of several databases, nodes are put into groups, and groups are linked together for synchronization. A trigger captures data for a table, which can include conditions and criteria for subsets. Triggers are grouped into channels for prioritization and control of data flow.

Figure 3.1. Configuration Data Model

3.1.1. Node Table

An instance of SymmetricDS that synchronizes data with one or more nodes. Each node has a unique identifier (*nodeId*) that is used when communicating, as well as a domain-specific identifier (*externalId*) that provides context within the local system.

Table 3.1. node

Column Name	Type	Null	Key	Default	Description
<i>node_id</i>	varchar(50)	N	PK		Unique identifier for a node
<i>node_group_id</i>	varchar(50)	N	FK		The node group that this node belongs to, such as "store"
<i>external_id</i>	varchar(50)	N			A domain-specific identifier for context within the local system. For example, the retail store number.
<i>sync_enabled</i>	booleanint	Y		0	Indicates whether this node should be sent synchronization. Disabled nodes are ignored by the triggers, so no entries are made in DataEvent for the node.
<i>sync_url</i>	varchar(2000)	Y			The URL to contact the node for synchronization.
<i>schema_version</i>	varchar(50)	Y			The version of the database schema this node manages. Useful for specifying synchronization by version.
<i>symmetric_version</i>	varchar(50)	Y			The version of SymmetricDS running at this node.

Column Name	Type	Null	Key	Default	Description
database_type	varchar(50)	Y			The database product name at this node as reported by JDBC.
database_version	varchar(50)	Y			The database product version at this node as reported by JDBC.
heartbeat_time	timestamp	Y			The last timestamp when the node sent a heartbeat, which is attempted every ten minutes by default.
timezone_offset	varchar(6)	Y			The timezone offset in RFC822 format at the time of the last heartbeat.
created_at_node_id	varchar(50)	Y			The id of the node where this node was created. This is filled with the id found in node_identity where registration was opened for the node.

3.1.2. Node Security Table

Security features like node passwords and open registration flag are stored in the NodeSecurity table.

Table 3.2. node_security

Column Name	Type	Null	Key	Default	Description
node_id	varchar(50)	N	PK FK		Unique identifier for a node
password	varchar(50)	N			The password used by the node to prove its identity during synchronization.
registration_enabled	booleanint	N		0	Indicates whether registration is open for this node.
registration_time	timestamp	Y			The timestamp when this node was registered.
initial_load_enabled	booleanint	N		0	Indicates whether an initial load will be sent to this node.
initial_load_time	timestamp	Y			The timestamp when this node started the initial load.
created_at_node_id	varchar(50)	Y			The id of the node where this node was created. This is filled with the id found in node_identity where registration was opened for the node.

3.1.3. Node Identity Table

After registration, this table will have one row representing the identity of the node. For a root node, the row is entered by the user.

Table 3.3. node_identity

Column Name	Type	Null	Key	Default	Description
node_id	varchar(50)	N	PK FK		Unique identifier for a node

3.1.4. Node Group Table

A category of Nodes that synchronizes data with one or more NodeGroups. A common use of NodeGroup is to describe a level in a hierarchy of data synchronization.

Table 3.4. node_group

Column Name	Type	Null	Key	Default	Description
node_group_id	varchar(50)	N	PK		Unique identifier for a node group, usually named something meaningful, like "store" or "warehouse".
description	varchar(50)	Y			A description of this node group.

3.1.5. Node Group Link Table

A source NodeGroup sends its data updates to a target NodeGroup using a pull, push, or custom technique.

Table 3.5. node_group_link

Column Name	Type	Null	Key	Default	Description
source_node_group_id	varchar(50)	N	PK FK		The node group where data changes should be captured.
target_node_group_id	varchar(50)	N	PK FK		The node group where data changes will be sent.
data_event_action	char(1)	N		W	The notification scheme used to send data changes to the target node group. (P = Push, W = Wait for Pull)

3.1.6. Channel Table

A category of data that can be synchronized independently of other Channels. Channels allow control over the type of data flowing and prevents one type of synchronization from contending with another.

Table 3.6. channel

Column Name	Type	Null	Key	Default	Description
channel_id	varchar(50)	N	PK		A unique identifier, usually named something meaningful, like "sales" or "inventory".
processing_order	integer	N		1	Order of sequence to process channel data.
max_batch_size	integer	N		1000	The maximum number of Data Events to process within a batch for this channel.
max_batch_to_send	integer	N		60	The maximum number of batches to send during a synchronization between two nodes.
enabled	booleanint	N		1	Indicates whether channel is enabled or not.
description	varchar(1000)	Y			Description on the type of data carried in this channel.

3.1.7. Node Channel Control Table

The Node Channel Control is used to ignore or suspend a channel. A channel that is ignored will have its Data Events batched and they will immediately be marked as "OK" without sending them. A channel that is suspended is skipped when batching Data Events.

Table 3.7. node_channel_ctl

Column Name	Type	Null	Key	Default	Description
node_id	varchar(50)	N	PK FK		Unique identifier for a node
channel_id	varchar(50)	N	PK FK		A unique identifier, usually named something meaningful, like "sales" or "inventory".
suspend_enabled	booleanint	Y		0	Indicates if this channel is suspended, which prevents its Data Events from being batched.
ignore_enabled	booleanint	Y		0	Indicates if this channel is ignored, which marks its Data Events as if they were actually processed.

3.1.8. Trigger Table

The database triggers that capture changes in the database are automatically generated by SymmetricDS. Configuration of which triggers are generated and how they will behave is stored in the Trigger table.

Table 3.8. trigger

Column Name	Type	Null	Key	Default	Description
trigger_id	integer	N	PK		Unique identifier for a trigger.
source_schema_name	varchar(50)	Y			The schema name where the source table resides. The database trigger will be created for the source_table_name

Column Name	Type	Null	Key	Default	Description
					in this schema.
source_table_name	varchar(50)	N			The name of the source table that will have a trigger installed to watch for data changes.
target_schema_name	varchar(50)	Y			The schema name where the target table resides. When data is sent to the remote database, the SQL statements used to load the data will qualify the target_table_name with this schema.
target_table_name	varchar(50)	Y			The name of the target table that will have data changes synchronized to it.
source_node_group_id	varchar(50)	N			The node group that will install this trigger to watch for data changes,
target_node_group_id	varchar(50)	N			The node group that will have data changes synchronized to it.
channel_id	varchar(50)	N			The channel that data changes will flow through.
sync_on_update	booleanint	N		1	Whether or not to install an update trigger.
sync_on_insert	booleanint	N		1	Whether or not to install an insert trigger.
sync_on_delete	booleanint	N		1	Whether or not to install a delete trigger.
sync_on_incoming_batch	booleanint	N		0	Whether or not an incoming batch that loads data into this table should cause the triggers to capture Data Events. Be careful turning this on, because an update loop is possible.
sync_column_level	booleanint	N		0	Perform column level synchronization for updates, so only the fields that changed are updated on the target database. When updates come from multiple sources at the same time, this enables merging of changes that do not conflict. The entire change row is still sent, which preserves the ability to fallback to an insert if the row does not exist.
name_for_update_trigger	varchar(30)	Y			Override the default generated name for the update trigger.
name_for_insert_trigger	varchar(30)	Y			Override the default generated name for the insert trigger.
name_for_delete_trigger	varchar(30)	Y			Override the default generated name for the delete trigger.
sync_on_update_condition	varchar(1000)	Y			Specify a condition for the update trigger firing using an expression specific to the database.
sync_on_insert_condition	varchar(1000)	Y			Specify a condition for the insert trigger firing using an expression specific to the database.
sync_on_delete_condition	varchar(1000)	Y			Specify a condition for the delete trigger firing using an expression specific to the database.
initial_load_select	varchar(1000)	Y			Specify a where-clause for an initial load of this table. The table is aliased as "t". Replacement variables are \$(nodeId), \$(groupId), and \$(externalId).
node_select	varchar(1000)	Y			Specify a where-clause for selecting the nodes that will receive data changes. The node table is aliased as "c".
tx_id_expression	varchar(1000)	Y			Override the default expression for the transaction identifier that groups the data changes that were committed together.

Column Name	Type	Null	Key	Default	Description
excluded_column_names	varchar(1000)	Y			Specify a comma-delimited list of columns that should not be synchronized from this table.
initial_load_order	integer	N		1	Order sequence of this table when an initial load is sent to a node.
create_time	timestamp	N			Timestamp when this entry was created.
inactive_time	timestamp	Y			Timestamp when this entry was inactivated, which stops capturing of data changes.
last_updated_by	varchar(50)	Y			The user who last updated this entry.
last_updated_time	timestamp	N			Timestamp when a user last updated this entry.

3.1.9. Parameter Table

The Parameter table provides a way to manage most SymmetricDS settings in the database.

Table 3.9. parameter

Column Name	Type	Null	Key	Default	Description
external_id	varchar(50)	N	PK		Target the parameter at a specific external id. To target all nodes, use the value of 'ALL.'
node_group_id	varchar(50)	N	PK FK		Target the parameter at a specific node group id. To target all groups, use the value of 'ALL.'
param_key	varchar(100)	N	PK		The name of the parameter.
param_value	varchar(1000)	N			The value of the parameter.

3.1.10. Registration Redirect Table

The Registration redirect table provides a way for a centralized registration server to redirect registering Nodes to their prospective parent Node in a multi-tiered deployment.

Table 3.10. registration_redirect

Column Name	Type	Null	Key	Default	Description
registrant_external_id	varchar(50)	N	PK		Maps the external id of a registration request to a different parent node.
registration_node_id	varchar(50)	N	PK		The id of the Node that a registration request should be redirected to.

3.2. Runtime Data Model

At runtime, the configuration is used to capture data changes and route them to nodes. The data changes are placed together in a single unit called a batch that can be loaded by another node. Outgoing batches are delivered to nodes and acknowledged. Incoming batches are received and loaded. History is recorded for batch status changes and statistics.

Figure 3.2. Runtime Data Model

3.2.1. Trigger History Table

A history of a table's definition and the Trigger used to capture data from the table. When a database trigger captures a data change, it references a Trigger Hist entry so it is possible to know which columns the data represents. Trigger Hist entries are made during the sync trigger process, which runs at each startup, each night in the SyncTriggersJob, or any time the syncTriggers() JMX method is manually invoked. A new entry is made when a table definition or a Trigger definition is changed, which causes a database trigger to be created or rebuilt.

Table 3.11. trigger_hist

Column Name	Type	Null	Key	Default	Description
trigger_hist_id	integer	N	PK		Unique identifier for a Trigger Hist.
trigger_id	integer	N	FK		Unique identifier for a Trigger.
source_table_name	varchar(50)	N			The name of the source table that will have a trigger installed to watch for data changes.
source_catalog_name	varchar(50)	Y			The catalog name where the source table resides.
source_schema_name	varchar(50)	Y			The schema name where the source table resides.
name_for_insert_trigger	varchar(50)	N			The name used when the insert trigger was created.
name_for_update_trigger	varchar(50)	N			The name used when the update trigger was created.
name_for_delete_trigger	varchar(50)	N			The name used when the delete trigger was created.
table_hash	integer	N			A hash of the table definition, used to detect changes in the definition.
column_names	longvarchar	N			The column names defined on the table. The column names are stored in comma-separated values (CSV) format.
pk_column_names	longvarchar	N			The primary key column names defined on the table. The column names are stored in comma-separated values (CSV) format.
last_trigger_build_reason	char(1)	N			The following reasons for a change are possible: New

Column Name	Type	Null	Key	Default	Description
					trigger that has not been created before (N); Schema changes in the table were detected (S); Configuration changes in Trigger (C); Trigger was missing (T).
create_time	timestamp	N			The date and time when this entry was recorded.
inactive_time	timestamp	Y			The date and time when a Trigger was inactivated.

3.2.2. Data Table

The captured data change that occurred to a row in the database. Entries in Data are created by database triggers.

Table 3.12. data

Column Name	Type	Null	Key	Default	Description
data_id	integer	N	PK		Unique identifier for a data
table_name	varchar(50)	N			The name of the table in which a change occurred that this entry records.
event_type	char(1)	Y			The type of event captured by this entry. For triggers, this is the change that occurred, which is "I" for insert, "U" for update, or "D" for delete. Other events include: "R" for reloading the entire table (or subset of the table) to the node; "S" for running dynamic SQL at the node, which is used for adhoc administration.
row_data	longvarchar	Y			The captured data change from the synchronized table. The column values are stored in comma-separated values (CSV) format.
pk_data	longvarchar	Y			The primary key values of the captured data change from the synchronized table. This data is captured for updates and deletes. The primary key values are stored in comma-separated values (CSV) format.
trigger_hist_id	integer	N	FK		The foreign key to the Trigger Hist entry that contains the primary key and column names for the table being synchronized.
create_time	timestamp	Y			The timestamp when the data change was captured.

3.2.3. Data Event Table

The Data Event represents routing of a Data to one or more Nodes. Entries in Data Event are created by database triggers.

Table 3.13. data_event

Column Name	Type	Null	Key	Default	Description
data_id	integer	N	PK FK		The Data that will be routed.
node_id	varchar(50)	N	PK FK		The Node that will receive the Data.
channel_id	varchar(50)	N	FK		The channel that this data belongs to, such as "prices"
transaction_id	varchar(1000)	Y			An optional transaction identifier that links multiple data changes together as the same transaction.
batch_id	integer	Y	FK		A unique identifier for a batch that will be a unit of delivery for multiple Data Events.
batched	char(1)	N		0	Whether or not this Data Event is prepared for a batch.

3.2.4. Outgoing Batch Table

The Outgoing Batch is used for tracking the sending a collection of Data to a Node in the system. A new Outgoing Batch is created by the Outgoing Batch Service and given a status of "NE". After sending the Outgoing Batch to its target Node, the status becomes "SE". The Node responds with either a success status of "OK" or an error status of "ER". An error while sending to the Node also results in an error status of "ER" regardless of whether the Node sends that acknowledgement.

Table 3.14. outgoing_batch

Column Name	Type	Null	Key	Default	Description
batch_id	integer	N	PK		A unique ID for the Batch.
node_id	varchar(50)	Y			The Node that will be sent this Batch.
channel_id	varchar(50)	Y			The Channel that categorizes the Data in this Batch.
batch_type	char(2)	N		EV	Batch types include events from triggers when rows change (EV) and initial loads of data that send an entire table (IL).
status	char(2)	Y			The current status of the Batch can be newly created (NE), sent to a Node (SE), acknowledged as successful (OK), and error (ER).
create_time	timestamp	Y			The date and time when the Batch was created.

3.2.5. Outgoing Batch History Table

A history of status changes to the Outgoing Batch, along with statistics of the work performed.

Table 3.15. outgoing_batch_hist

Column Name	Type	Null	Key	Default	Description
batch_id	integer	N	PK		A unique ID for the Batch.
node_id	varchar(50)	Y			The Node that will be sent this Batch.
status	char(2)	Y			The current status of the Batch can be newly created (NE), sent to a Node (SE), acknowledged as successful (OK), or error (ER).
start_time	timestamp	Y			The date and time when the process for this entry was started. (1.4)
end_time	timestamp	Y			The date and time when the process for this entry was ended. (1.4)
data_event_count	integer	Y			The number of Data Events in the Batch. This is only populated for a new status (NE).
failed_data_id	integer	Y			For a status of error (ER), this is the Data entry that was being processed when the Batch failed.
hostname	varchar(50)	Y			The name of the machine processing the Batch, which is meaningful when running a cluster of SymmetricDS instances. (1.4)
network_millis	integer	Y			The amount of time in milliseconds spent using the network to send the Batch. This is currently unimplemented. (1.4)
filter_millis	integer	Y			The amount of time in milliseconds spent in filters. This is currently unimplemented. (1.4)
database_millis	integer	Y			The amount of time in milliseconds spent in the database. This is only populated for a status of new (NE). (1.4)
sql_state	varchar(10)	Y			For a status of error (ER), this is the XOPEN or SQL 99 SQL State. (1.4)
sql_code	integer	Y			For a status of error (ER), this is the error code from the database that is specific to the vendor. (1.4)
sql_message	varchar(50)	Y			For a status of error (ER), this is the error message that describes the error. (1.4)

3.2.6. Incoming Batch Table

The Incoming Batch is used for tracking the status of loading an Outgoing Batch from another Node. Data is loaded and committed at the batch level. The status of the Incoming Batch is either successful (OK) or error (ER).

Table 3.16. incoming_batch

Column Name	Type	Null	Key	Default	Description
batch_id	integer	N	PK		A unique ID for the Batch.
node_id	varchar(50)	N	PK		The Node that sent this Batch.
status	char(2)	Y			The current status of the Batch can be successfully loaded (OK) or error (ER).
create_time	timestamp	Y			The date and time when the Batch was first received.

3.2.7. Incoming Batch History Table

A history of status changes to the Incoming Batch, along with statistics of the work performed.

Table 3.17. incoming_batch_hist

Column Name	Type	Null	Key	Default	Description
batch_id	integer	N	PK		A unique ID for the Batch.
node_id	varchar(50)	Y			The Node that sent this Batch to be loaded.
status	char(2)	Y			The current status of the Batch can be loaded successfully (OK), skipped because previously loaded (SK), or error (ER).
start_time	timestamp	Y			The date and time when the load for this entry was started.
end_time	timestamp	Y			The date and time when the load for this entry was ended.
failed_row_number	integer	Y			For a status of error (ER), this is the row number (starting with row one) that was being processed when the Batch failed.
hostname	varchar(50)	Y			The name of the machine processing the Batch, which is meaningful when running a cluster of SymmetricDS instances.
network_millis	integer	Y			The amount of time in milliseconds spent using the network to receive the Batch. This is currently unimplemented.
filter_millis	integer	Y			The amount of time in milliseconds spent in filters. This currently includes time from IDataLoaderFilters.
database_millis	integer	Y			The amount of time in milliseconds spent in the database.
sql_state	varchar(10)	Y			For a status of error (ER), this is the XOPEN or SQL 99 SQL State. (1.4)
sql_code	integer	Y			For a status of error (ER), this is the error code from the database that is specific to the vendor. (1.4)
sql_message	varchar(50)	Y			For a status of error (ER), this is the error message that describes the error. (1.4)

Chapter 4. Architecture

The SymmetricDS software allows for outgoing and incoming changes to be synchronized to/from other databases. The Node that initiates the connection is the client, and the Node receiving a connection is the host. Because synchronization is configurable to push or pull in either direction, the same Node can act as either a client or a host in different circumstances.

The software consists of a series of jobs, managers, servlets, and services wired together through dependency injection using the [Spring Framework](#).

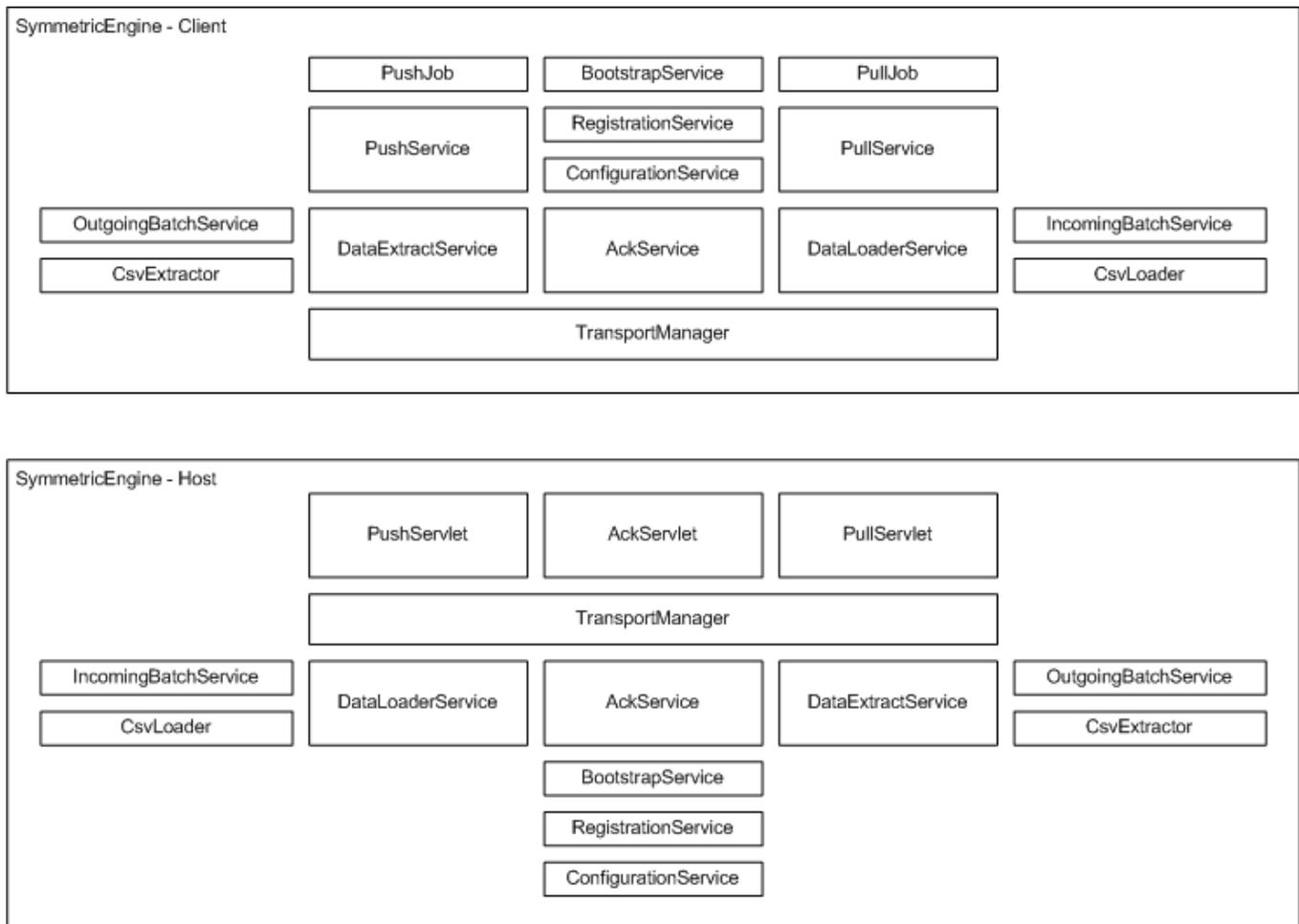


Figure 4.1. Software Stack

As a client, the Node runs the Push Job and Pull Job on a timer thread in order to synchronize with a host Node. The Push Job uses services to batch, extract, and stream data to another Node (i.e. it pushes data). The response from a push is a list of batch acknowledgements to indicate that data was loaded. The Pull Job uses services to load data that is streamed from another Node (i.e. it pulls data). After loading data, a second connection is made to send a list of batch acknowledgements.

As a host, the Node waits for incoming connections that pull, push, or acknowledge data changes. The

Push Servlet uses services to load data that is pushed from a client Node. After loading data, it responds with a list of batch acknowledgements. The Pull Servlet uses services to batch, extract, and stream data back to the client Node. The Ack Servlet uses services to update the status of data that was loaded at a client Node.

The Transport Manager handles the incoming and outgoing streams of data between Nodes. The default transport is based on a simple implementation over HTTP. An internal transport is also provided. It is possible to add other implementations, such as a socket-based transport manager.

Node communication over HTTP is represented in the following figure.

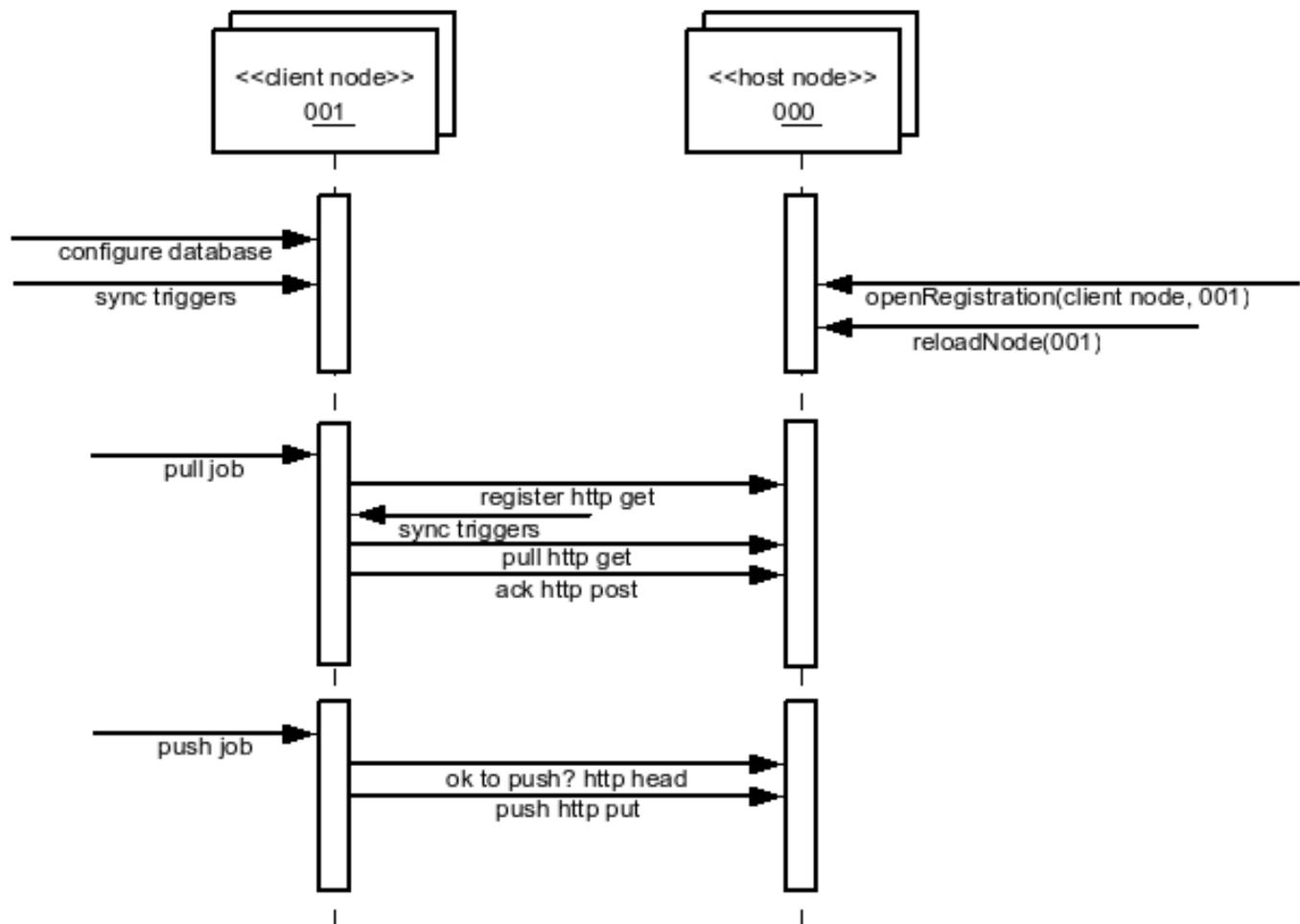


Figure 4.2. Node Communication

The `SymmetricEngine` is wrapper API that can be used to directly start the client services only. The `SymmetricWebServer` is a wrapper API that can be used to directly start *both* the client and host services inside a Jetty web container.

4.1. Deployment Options

The following deployment options are possible:

- Web application archive (WAR) deployed to an application server

This option means packaging a WAR file and deploying to your favorite web server, like Apache Tomcat. It's a little more work, but you can configure the web server to do whatever you need.

- Standalone service that embeds Jetty web server

This option means running the `sym` command line, which launches the built-in Jetty web server. This is a simple option because it is already provided, but you lose the flexibility to configure the web server any further.

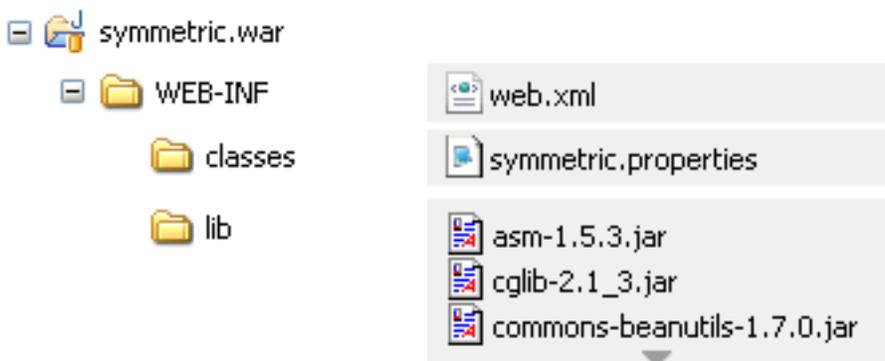
- Embedded as a Java library in an application

This option means you must write a wrapper Java program that runs SymmetricDS. You would probably use Jetty web server, which is also embeddable. You could bring up an embedded database like Derby. You could configure the web server, database, or SymmetricDS to do whatever you needed, but it's also the most work of the 3 options.

The deployment model you choose depends on how much flexibility you need versus how easy you want it to be. Both Jetty and Tomcat are excellent, scalable web servers that compete with each other and have great performance. Most people choose either the *Standalone* or *Web Archive* with Tomcat 5.5 or 6. Deploying to Tomcat is a good middle-of-the-road decision that requires a little more work for more flexibility.

4.1.1. Web Archive

As a web application archive, a WAR or EAR file is deployed to an application server, such as Tomcat, Jetty, or JBoss. The structure of the archive should have a `web.xml` file in the `WEB-INF` folder, the `symmetric.properties` file in the `WEB-INF/classes` folder, and the required JAR files in the `WEB-INF/lib` folder.



The `WEB-INF/web.xml` file is configured with a `SymmetricEngineContextLoaderListener` the required `SymmetricFilter` mapping, and the required `SymmetricServlet` mapping. Note that this was changed in version 1.4.0 to make it easier to configure Symmetric.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  version="2.5">

  <display-name>sync</display-name>

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <!-- You can optionally specify other Spring files to load into same context here -->
    <param-value>classpath:symmetric.xml</param-value>
  </context-param>

  <filter>
    <filter-name>SymmetricFilter</filter-name>
    <filter-class>
      org.jumpmind.symmetric.web.SymmetricFilter
    </filter-class>
  </filter>

  <filter-mapping>
    <filter-name>SymmetricFilter</filter-name>
    <servlet-name>/*</servlet-name>
  </filter-mapping>

  <listener>
    <listener-class>
      org.jumpmind.symmetric.SymmetricEngineContextLoaderListener
    </listener-class>
  </listener>

  <servlet>
    <servlet-name>SymmetricServlet</servlet-name>
    <servlet-class>
      org.jumpmind.symmetric.web.SymmetricServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>SymmetricServlet</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>

</web-app>

```

This example starts all the SymmetricDS Servlets with Filters to compress the stream, authenticate nodes, and reject nodes when the server is too busy.

The `web.base.servlet.path` property in `symmetric.properties` can be set if the `SymmetricServlet` needs to coexist with other Servlets.

4.1.2. Standalone

A standalone service can use the `sym` command line options to start a server. An embedded instance of Jetty is used to service web requests for all the servlets.

```
/symmetric/bin/sym --properties root.properties --port 8080 --server
```

This example starts the SymmetricDS server on port 8080 with the startup properties found in the `root.properties` file.

4.1.3. Embedded

A Java application with the SymmetricDS Java Archive (JAR) library on its classpath can use the `SymmetricEngine` to start the server.

```
import org.jumpmind.symmetric.SymmetricEngine;

public class StartSymmetricDSEngine {

    public static void main(String[] args) throws Exception {
        String workingDirectory = System.getProperty("user.dir");

        SymmetricWebServer node = new SymmetricWebServer(new SymmetricEngine("classpath://my-
            + workingDirectory + "/my-environment.properties"));

        // this will create the database, sync triggers, start jobs running
        node.start(8080);

        // this will stop the node
        node.stop();
    }
}
```

This example starts the SymmetricDS server on port 8080 with startup properties found in two locations. The first file, `my-application.properties`, is packaged in the application to provide properties that override the SymmetricDS default values. The second file, `my-environment.properties`, is located in a working directory that overrides properties specific to the environment. This allows the same application to deploy to development and production environments using different `my-environment.properties`.

Chapter 5. Basic Configuration

To get an instance of SymmetricDS running, it needs to be given an identity and it needs to know how to connect to the database it will manage. A basic way to specify this is to place properties in the `symmetric.properties` file. When started up, the SymmetricDS reads the configuration and state from the database. If the configuration tables are missing, they are created automatically (auto creation can be disabled). A basic configuration describes the following:

- **Node Group** - each Node belongs to a group
- **Node Group Link** - two Nodes Groups are linked together for synchronization
- **Node** - an instance of SymmetricDS
- **Channel** - data is categorized to synchronize independently
- **Trigger** - specify which changes in the database are captured

During start up, the Triggers are verified against the database, and database triggers are installed on tables that require data changes to be captured. The PullJob and PushJob begin running as required to synchronize changes with other Nodes.

5.1. Basic Properties

Each Node requires properties that will connect it to the database and register it with a parent Node. To give a Node its identity, the following properties are used:

group.id

The Node Group that this Node is a member of. Synchronization is specified between Node Groups, which means you only need to specify it once for multiple Nodes in the same group. For example, you might have groups of "STORE", "REGION", and "CENTRAL" that synchronize.

external.id

The External ID for this Node has meaning to the user and provides integration into the system where it is deployed. For example, it might be a retail store number or a region number. The External ID can be used in expressions for conditional and subset data synchronization. Behind the scenes, each Node has a unique sequence number for tracking synchronization events. That makes it possible to assign the same External ID to multiple Nodes, if desired.

my.url

The URL where this Node can be contacted for synchronization. At startup and during each heartbeat, the Node updates its entry in the database with this URL.

When a new Node is first started, it has no information about synchronizing. It contacts the registration server in order to join the network and receive its configuration. The configuration for all Nodes is stored on the registration server, and the URL must be specified in the following property:

registration.url

The URL where this Node can connect for registration to receive its configuration. The registration server is part of SymmetricDS and is enabled as part of the deployment.

For a deployment to an application server, it is common for database connection pools to be found in the Java naming directory (JNDI). In this case, set the following property:

db.jndi.name

The name of the database connection pool to use, which is registered in the JNDI directory tree of the application server. It is recommended that this DataSource is NOT transactional, because SymmetricDS will handle its own transactions.

For a deployment where the database connection pool should be created using a JDBC driver, set the following properties:

db.driver

The class name of the JDBC driver.

db.url

The JDBC URL used to connect to the database.

db.user

The database username, which is used to login, create, and update SymmetricDS tables.

db.password

The password for the database user.

5.2. Node Group

Each Node must belong to a Node Group, a collection of one or more Nodes. A common use of Node Groups is to describe a level in a hierarchy of data synchronization. For example, at a retail store chain, there might be a few Nodes that belong to "corp", which sync with hundreds of Nodes that belong to "store", which sync with thousands of Nodes that belong to "register".

The following SQL statements would create Node Groups for "corp" and "store".

```
insert into SYM_NODE_GROUP
  (node_group_id, description)
values
  ('store', 'A retail store node');

insert into SYM_NODE_GROUP
  (node_group_id, description)
values
  ('corp', 'A corporate node');
```

5.3. Node Group Link

To establish synchronization between Nodes, two Node Groups are linked together. The direction of synchronization is determined by specifying a source and target Node Group. If synchronization should occur in both directions, then two links are created in opposite directions. The target Node Group receives data changes by either push or pull methods. A push method causes the source Node Group to connect to the target, while a pull method causes it to wait for the target to connect to it.

The following SQL statements links the "corp" and "store" Node Groups for synchronization. It configures the "store" Nodes to push their data changes to the "corp" Nodes, and the "corp" Nodes to send changes to "store" Nodes by waiting for a pull.

```
insert into SYM_NODE_GROUP_LINK
  (source_node_group, target_node_group, data_event_action)
values
  ('store', 'corp', 'P');

insert into SYM_NODE_GROUP_LINK
  (source_node_group, target_node_group, data_event_action)
values
  ('corp', 'store', 'W');
```

5.4. Node

Each instance of SymmetricDS is a Node that can be uniquely identified. The Node has a unique identifier used by the system, and the user provides an external identifier for context in the local system. For most common use, the two identifiers are the same. The registration process generates and sends the identity and password to the Node, along with its synchronization configuration. The top-level registration server must have its identity provided by the user since it has no parent to contact.

The following SQL statements setup a top-level registration server as a Node identified as "00000" in the "corp" Node Group.

```
insert into SYM_NODE
  (node_id, node_group_id, external_id, sync_enabled)
values
  ('00000', 'corp', '00000', 1);

insert into SYM_NODE_IDENTITY values ('00000');
```

5.5. Channel

Data changes in the database are captured in the order that they occur, which is preserved when synchronizing to other Nodes. Some data may need priority for synchronization despite the normal order of events. Channels provide a higher-level processing order of data, a limit on the amount of data, and isolation from errors in other channels. By categorizing data into channels and assigning them to Triggers, the user gains more control and visibility into the flow of data.

The following SQL statements setup channels for a retail store. An "item" channel includes data for items and their prices, while a "sale_transaction" channel includes data for ringing sales at a register.

```
insert into SYM_CHANNEL
  (channel_id, processing_order, max_batch_size, enabled, description)
values
  ('item', 10, 1000, 1, 'Item and pricing data');

insert into SYM_CHANNEL
  (channel_id, processing_order, max_batch_size, enabled, description)
values
  ('sale_transaction', 1, 1000, 1, 'retail sale transactions from register');
```

5.6. Trigger

At the heart of SymmetricDS are Triggers that define what data to capture. Nodes in the source Node Group will capture changes for a table and send them to a target Node Group. Changes can include inserts, updates, or deletes to the table, and it is even possible to filter data by a conditional expression. An entry in Trigger results in a database trigger being installed on the table. Whenever the Trigger entry is updated, the `last_updated_time` should be updated to indicate that the database trigger should also be updated.

The following SQL statement defines a Trigger that will capture data for a table named "item" whenever data is inserted, updated, or deleted. Data will be captured on Nodes in the "corp" Node Group and sent to Nodes in the "store" Node Group.

```
insert into SYM_TRIGGER
  (source_table_name, source_node_group_id, target_node_group_id, channel_id,
  sync_on_insert, sync_on_update, sync_on_delete,
  initial_load_order, last_updated_by, last_updated_time, create_time)
values
  ('item', 'corp', 'store', 'item',
  1, 1, 1,
  105, 'demo', current_timestamp, current_timestamp);
```

5.7. Using Sync Conditions

Sync Conditions can be specified for capturing the changes to rows in a table. If the condition is met, the data change is captured for sending to other Nodes. If the condition is not met, the data change is allowed, but it will not be captured and sent to other Nodes. The condition runs within the context of the database trigger event, and it uses the language specific to the database platform. A condition can be specified for insert, update, and delete events, and it is run for each row of that event.

Because a condition makes a decision about whether or not to capture a data change, it effectively creates a subset of the table for synchronization to all Nodes in the Node Group. If a different subset of the table is required for each Node, then see the section on Using Node Select.

The following SQL statement defines a Trigger that will capture data for a table named "sale_transaction" only when the workstation column has a value of "001". All Nodes in the "store" Node Group will receive a subset of the table for only that workstation.

```
insert into SYM_TRIGGER
  (source_table_name, source_node_group_id, target_node_group_id, channel_id,
  sync_on_insert, sync_on_update, sync_on_delete,
  sync_on_insert_condition, sync_on_update_condition, sync_on_delete_condition,
  initial_load_order, last_updated_by, last_updated_time, create_time)
values
  ('sale_transaction', 'corp', 'store', 'sale_transaction',
  1, 1, 1,
  'workstation = ''001''', 'workstation = ''001''', 'workstation = ''001''',
  105, 'demo', current_timestamp, current_timestamp);
```

5.8. Using Node Select

After a data change is captured, the Nodes which will receive the change are selected. By default, all Nodes in the target Node Group for the Trigger will be selected. To limit which Nodes are selected, a condition is specified for the node_select of the Trigger. The condition is written in SQL syntax that is appended to the "where" clause of a "select" statement on the Node table. The table is aliased as "c" in order to allow a sub-select to correlate a join. The condition is run within the context of the database trigger event, so it can access the row being changed.

Because the node_select makes a decision about which Nodes receive the data change, it effectively creates a subset of the table for different Nodes. If the same subset of the table is required for all Nodes in the Node Group, then see the section on Using Sync Conditions.

The following variables can be used in the node_select, which will be replaced when the database trigger is built:

- \$(oldTriggerValue) - alias for columns on the "before" row that was updated or deleted
- \$(newTriggerValue) - alias for columns on the "after" row that was updated or inserted
- \$(curTriggerValue) - alias for columns on the current row that was changed

These variables are replaced with syntax in the language of the database being used. For example, on Oracle the "\$(newTriggerValue)" is replaced with ":new". The curTriggerValue will use the oldTriggerValue on a delete change, otherwise use newTriggerValue.

The following SQL statement defines a Trigger that will capture all data changes for a table named "sale_transaction", and select Nodes to receive changes by using the value of the "store" column. The Nodes that have been set with an external ID that matches the "store" column will receive only those rows.

```
insert into SYM_TRIGGER
  (source_table_name, source_node_group_id, target_node_group_id, channel_id,
  sync_on_insert, sync_on_update, sync_on_delete,
```

```

node_select,
initial_load_order, last_updated_by, last_updated_time, create_time)
values
('sale_transaction', 'corp', 'store', 'sale_transaction',
1, 1, 1,
'and external_id = $(curTriggerValue).store',
105, 'demo', current_timestamp, current_timestamp);

```

5.9. Using Initial Load Select

The Initial Load sends all required rows for each table to a Node, which prepares its database with an initial synchronized state. Afterwards, SymmetricDS begins sending incremental changes as data is changed. By default, the initial load will send all rows of a table. To send a subset of data, a condition is specified for the `initial_load_select` of the Trigger. The condition is written in SQL syntax that is appended to the "where" clause of a "select" statement on the table. The table is aliased as "t" in order to allow a sub-select to correlate a join.

It's important that the condition used for the initial state of a table matches the condition for capturing and sending changes. In other words, the condition used for the `initial_load_select` should be consistent with any Sync Conditions or Node Select Conditions, in order to guarantee consistent data in the table.

The following variables can be used in the `initial_load_select`, which will be replaced at runtime:

- `$(nodeId)` - the node ID of the target node
- `$(groupId)` - the node group ID of the target node
- `$(externalId)` - the external ID of the target node

The following SQL statement defines a Trigger for the "sale_transaction" table that will only synchronize the subset of data where the workstation is "001".

```

insert into SYM_TRIGGER
(source_table_name, source_node_group_id, target_node_group_id, channel_id,
sync_on_insert, sync_on_update, sync_on_delete,
sync_on_insert_condition, sync_on_update_condition, sync_on_delete_condition,
initial_load_select,
initial_load_order, last_updated_by, last_updated_time, create_time)
values
('sale_transaction', 'corp', 'store', 'sale_transaction',
1, 1, 1,
'workstation = ''001'', 'workstation = ''001'', 'workstation = ''001'',
'workstation = ''001'',
105, 'demo', current_timestamp, current_timestamp);

```

Chapter 6. Advanced Configuration

6.1. Multi-Tiered Synchronization

At times, there may be scenarios where data needs to flow through multiple tiers of nodes that are organized in a tree-like network with each tier requiring a different subset of data. For example, you may have a system where the lowest tier may be a computer or device located in a store. Those devices may connect to a server located physically at that store. Then the store server may communicate with a corporate server or even a regional server.

The different tiers would be device, store, region and corporate. A tier is typically represented by a node group. Each node in the tier would belong to the node group representing that tier.

A node will always push and pull data to other node groups according to the node group link configuration. A node can only pull and push data to other nodes that are represented as rows in the node table in its database. Because of this, a tree-like hierarchy of nodes can be created by having only a subset of nodes belonging to the same node group represented at the different branches of the tree.

If auto registration is turned *off*, then this setup must occur manually by opening registration for the desired nodes at the desired parent node and by configuring each node's `registration.url` to be the parent node's URL. The parent node is always tracked by the setting of the parent's `node_id` in the `created_at_node_id` column of the new node. When a node registers and downloads its configuration it is always provided the configuration for all nodes that were *created* at the node that it is registering with.

Most tiers will require the SymmetricDS *server* software. It must be deployed to any node that is pulled from or pushed to. If the node only pushes and pulls from other nodes, then only the client software (no HTTP server) is required.

Nodes that service many clients may be deployed as a web farm for scalability purposes. In this scenario a single node may have multiple SymmetricDS server instances that are configured the same and are fronted by some type of stateless load balancer.

6.2. Bi-Directional Synchronization

SymmetricDS allows tables to be synchronized bi-directionally. Note that an outgoing synchronization does not process changes during an incoming synchronization on the same node unless the trigger was created with the `sync_on_incoming_batch` flag set. If the `sync_on_incoming_batch` flag is set, then update loops are prevented by a feature that is available in most database dialects. During an incoming synchronization the source `node_id` is put into a database session variable that is available to the database trigger. Data events are not generated if the target `node_id` on an outgoing synchronization is equal to the source `node_id`.

When synchronizing tables in both directions, the `sync_column_level` flag may be enabled. This indicates to the data loader that only columns that have changed should be updated. In order to accomplish this, both old and new data are transmitted from the source node to the target node. Because of the extra data

overhead this feature should only be enabled if needed.

More complex conflict resolution strategies can be accomplished by using the `IDataLoaderFilter` extension point which has access to both old and new data.

6.3. Dead Triggers

Normally a Trigger is specified to capture data changes to a table and send them to a target Node Group. A dead Trigger is one that does not capture data changes. In other words, the `sync_on_insert`, `sync_on_update`, and `sync_on_delete` properties for the Trigger are all set to false. Because the Trigger is specified, it will be included in the initial load of data for target Nodes.

A dead Trigger might be used to load a read-only lookup table. It could be used to load a table that needs populated with example or default data. Another use is a recovery load of data for tables that have a single direction of synchronization. For example, a retail store records sales transaction that synchronize in one direction by trickling back to the central office. If the retail store needs to recover all the sales transactions, they can be sent are part of an initial load from the central office by setting up dead Triggers that "sync" in that direction.

The following SQL statement sets up a non-syncing dead Trigger that sends the `sale_transaction` table to the "store" Node Group from the "corp" Node Group during an initial load.

```
insert into SYM_TRIGGER
(source_table_name, source_node_group_id, target_node_group_id, channel_id,
sync_on_insert, sync_on_update, sync_on_delete,
initial_load_order, last_updated_by, last_updated_time, create_time)
values
('sale_transaction', 'corp', 'store', 'sale_transaction',
0, 0, 0,
105, 'demo', current_timestamp, current_timestamp);
```

6.4. Database Purging

6.5. Extension Points

SymmetricDS may be extended via a plug-in like architecture where extension point interfaces may be implemented by a custom class and registered with the synchronization engine. All supported extension points extend the `IExtensionPoint` interface. The currently available extension points are documented in the following sections.

When the synchronization engine starts up, a Spring post processor searches the Spring `ApplicationContext` for any registered classes which implement `IExtensionPoint`. An `IExtensionPoint` designates whether it should be auto registered or not. If the extension point is to be auto registered then the post processor registers the known interface with the appropriate service.

The `INodeGroupExtensionPoint` interface may be optionally implemented to designate that auto registered extension points should only be auto registered with specific node groups.

```
/**
 * Only apply this extension point to the 'root' node group.
 */
public String[] getNodeGroupIdsToApplyTo() {
    return new String[] { "root" };
}
```

SymmetricDS will look for Spring configured extensions in the application Classpath by importing any Spring XML configuration files found matching the following pattern:
META-INF/services/symmetric-*-ext.xml.

6.5.1. IParameterFilter

Parameter values can be specified in code using a parameter filter. Note that there can be only one parameter filter per engine instance. The `IParameterFilter` replaces the deprecated `IRuntimeConfig` from prior releases.

```
public class MyParameterFilter
    implements IParameterFilter, INodeGroupExtensionPoint {

    /**
     * Only apply this filter to stores
     */
    public String[] getNodeGroupIdsToApplyTo() {
        return new String[] { "store" };
    }

    public String filterParameter(String key, String value) {
        // look up a store number from an already existing properties file.
        if (key.equals(ParameterConstants.EXTERNAL_ID)) {
            return StoreProperties.getStoreProperties().
                getProperty(StoreProperties.STORE_NUMBER);
        }
        return value;
    }

    public boolean isAutoRegister() {
        return true;
    }

}
```

6.5.2. IDataLoaderFilter

Data can be filtered as it is loaded into the target database or when it is extracted from the source database.

As data is loaded into the target database, a filter can change the data in a column or save it somewhere else. It can also specify by the return type of the function call that the data loader should continue on and

load the data (by returning true) or ignore it (by returning false). One possible use of the filter might be to route credit card data to a secure database and blank it out as it loads into less-restricted reporting database.

An `IDataLoaderContext` is passed to each of the callback methods. A new context is created for each synchronization. The context provides methods to lookup column indexes by column name, get table meta data, and provides access to old data if the `sync_column_level` flag is enabled. The context also provides a means to share data during a synchronization between different rows of data that are committed in a database transaction and are in the same channel. It does so by providing a context cache which can be populated by the extension point.

Many times the `IDataLoaderFilter` will be combined with the `IBatchListener`. The `XmlPublisherFilter` (in the `org.jumpmind.symmetric.ext` package) is a good example of using the combination of the two extension points in order to create XML messages to be published to JMS.

A class implementing the `IDataLoaderFilter` interface is injected onto the `DataLoaderService` in order to receive callbacks when data is inserted, updated, or deleted.

```
public MyFilter implements IDataLoaderFilter {
    public boolean isAutoRegister() {
        return true;
    }

    public boolean filterInsert(IDataLoaderContext context,
        String[] columnValues) {
        return true;
    }

    public boolean filterUpdate(IDataLoaderContext context,
        String[] columnValues, String[] keyValues) {
        return true;
    }

    public void filterDelete(IDataLoaderContext context,
        String[] keyValues) {
        return true;
    }
}
```

The filter class is specified as a Spring-managed bean. A custom Spring XML file is specified as follows in a jar at `META-INF/services/symmetric-myfilter-ext.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <bean id="myFilter" class="com.mydomain.MyFilter"/>

</beans>
```

6.5.3. ITableColumnFilter

Implement this extension point to filter out specific columns from use by the dataloader. Only one column filter may be added per target table.

6.5.4. IBatchListener

This extension point is called whenever a batch has completed loading but before the transaction has committed.

6.5.5. IAcknowledgeEventListener

Implement this extension point to receive callback events when a batch is acknowledged. The callback for this listener happens at the point of extraction.

6.5.6. IReloadListener

Implement this extension point to listen in and take action before or after a reload is requested for a Node. The callback for this listener happens at the point of extraction.

6.5.7. IExtractorFilter

This extension point is called after data has been extracted, but before it has been streamed. It has the ability to inspect each row of data to take some action and indicate, if necessary, that the row should not be streamed.

6.5.8. ISyncUrlExtension

This extension point is used to select an appropriate URL based on the URI provided in the sync_url column of sym_node.

To use this extension point configure the sync_url for a node with the protocol of ext://beanName. The beanName is the name you give the extension point in the extension xml file.

6.5.9. INodeIdGenerator

This extension point allows SymmetricDS users to implement their own algorithms for how node ids and passwords are generated or selected during the registration process. There may be only one node generator per SymmetricDS instance.

6.5.10. ITriggerCreationListener

Implement this extension point to get status callbacks during trigger creation.

6.6. Encrypted Passwords

The `db.user` and `db.password` properties will accept encrypted text, which protects against casual observation. The text is prefixed with "enc:" to indicate that it is encrypted. To encrypt text, use the following command:

```
sym -e secret
```

The text is encrypted by the cipher defined as alias "sym.secret" in the Java keystore. The keystore is specified by the "sym.keystore.file" system property, which defaults to `security/keystore`. If a cipher is not found, a default cipher using Triple DES with a random password is generated.

6.7. Secure Transport

By specifying the "https" protocol for a URL, SymmetricDS will communicate over Secure Sockets Layer (SSL) for an encrypted transport. The following properties need to be set with "https" in the URL:

my.url

This is the URL of the current node, so if you want to force other nodes to communicate over SSL with this node, you specify "https" in the URL.

registration.url

This is the URL where the node will connect for registration when it first starts up. To protect the registration with SSL, you specify "https" in the URL.

For incoming HTTPS connections, SymmetricDS depends on the webserver where it is deployed, so the webserver must be configured for HTTPS. As a standalone deployment, the "sym" launcher command provides options for enabling HTTPS support.

6.7.1. Sym Launcher

The "sym" launch command uses Jetty as an embedded web server. Using command line options, the web server can be told to listen for HTTP, HTTPS, or both.

```
sym --port 8080 --server
```

```
sym --secure-port 8443 --secure-server
```

```
sym --port 8080 --secure-port 8443 --mixed-server
```

6.7.2. Tomcat

If you deploy SymmetricDS to Apache Tomcat, it can be secured by editing the

TOMCAT_HOME/conf/server.xml configuration file. There is already a line that can be uncommented and changed to the following:

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
  maxThreads="150" scheme="https" secure="true"
  clientAuth="false" sslProtocol="TLS"
  keystoreFile="/symmetric-ds-1.x.x/security/keystore" />
```

6.7.3. Keystores

When SymmetricDS connects to a URL with HTTPS, Java checks the validity of the certificate using the built-in trusted keystore located at JRE_HOME/lib/security/cacerts. The "sym" launcher command overrides the trusted keystore to use its own trusted keystore instead, which is located at security/cacerts. This keystore contains the certificate aliased as "sym" for use in testing and easing deployments. The trusted keystore can be overridden by specifying the "javax.net.ssl.trustStore" system property.

When SymmetricDS is run as a secure server with the "sym" launcher, it accepts incoming requests using the key installed in the keystore located at security/keystore. The default key is provided for convenience of testing, but should be re-generated for security.

6.7.4. Generating Keys

To generate new keys and install a server certificate, use the following steps:

1. Open a command prompt and navigate to the security subdirectory of your SymmetricDS installation.
2. Delete the old key pair and certificate.

```
keytool -keystore keystore -delete -alias sym
```

```
keytool -keystore cacerts -delete -alias sym
```

```
Enter keystore password:  changeit
```

3. Generate a new key pair.

```
keytool -keystore keystore -alias sym -genkey -keyalg RSA -validity 10950
```

```
Enter keystore password:  changeit
What is your first and last name?
  [Unknown]:  localhost
What is the name of your organizational unit?
  [Unknown]:  SymmetricDS
What is the name of your organization?
  [Unknown]:  JumpMind
What is the name of your City or Locality?
  [Unknown]:
What is the name of your State or Province?
```

```
[Unknown]:  
What is the two-letter country code for this unit?  
[Unknown]:  
Is CN=localhost, OU=SymmetricDS, O=JumpMind, L=Unknown, ST=Unknown, C=Unknown  
correct?  
[no]: yes  
  
Enter key password for <sym>  
      (RETURN if same as keystore password):
```

4. Export the certificate from the private keystore.

```
keytool -keystore keystore -export -alias sym -rfc -file sym.cer
```

5. Install the certificate in the trusted keystore.

```
keytool -keystore cacerts -import -alias sym -file sym.cer
```

Chapter 7. Administration

7.1. Changing Triggers

A trigger row may be updated using SQL to change a synchronization definition. SymmetricDS will look for changes each night or whenever the Sync Triggers Job is run (see next section). For example, a change to place the table `price_changes` into the price channel would be accomplished with the following statement:

```
update SYM_TRIGGER
set channel_id = 'price',
    last_updated_by = 'jsmith',
    last_updated_time = current_timestamp
where source_table_name = 'price_changes';
```

The Trigger entry can be inactivated, which would drop the triggers (and any functions used by them) from the database. For example, dropping all the triggers used by the client node group would be accomplished with the following statement:

```
update SYM_TRIGGER
set inactive_time = current_timestamp,
    last_updated_by = 'jsmith',
    last_updated_time = current_timestamp
where source_node_group_id = 'client';
```

All configuration should be managed centrally at the registration node. If enabled, configuration changes will be synchronized out to client nodes. When trigger changes reach the client nodes the Sync Triggers process will run automatically.

Centrally, the trigger changes will not take effect until the Sync Triggers Job runs. Instead of waiting for the Sync Triggers Job to run overnight after making a Trigger change, invoke the `syncTriggers()` method over JMX or simply restart the SymmetricDS server.

7.2. Changing Configuration

The configuration of the your system as defined in the `sym_*` tables may be modified at runtime. By default any changes made to the `sym_*` tables (with the exception of `sym_node`) should be made at the registration server. The changes will be synchronized out to the leaf nodes by SymmetricDS triggers that are automatically created on the tables.

If this behavior is not desired, the feature can be turned off using a parameter. Custom triggers may be added to the `sym_*` tables when the auto syncing feature is disabled.

7.3. Sync Triggers Job

SymmetricDS examines the current configuration, corresponding database triggers, and the underlying tables to determine if database triggers need created or updated. The change activity is recorded on the Trigger Hist table with a reason for the change. The following reasons for a change are possible:

- N - New trigger that has not been created before
- S - Schema changes in the table were detected
- C - Configuration changes in Trigger
- T - Trigger was missing

A configuration entry in Trigger without any history in Trigger Hist results in a new trigger being created (N). The Trigger Hist stores a hash of the underlying table, so any alteration to the table causes the trigger to be rebuilt (S). When the last_updated_time is changed on the Trigger entry, the configuration change causes the trigger to be rebuilt (C). If an entry in Trigger Hist is missing the corresponding database trigger, the trigger is created (T).

The process of examining triggers and rebuilding them is automatically run during startup and each night by the SyncTriggersJob. The user can also manually run the process at any time by invoking the syncTriggers() method over JMX. The SyncTriggersJob is enabled by default to run at 15 minutes past midnight. If SymmetricDS is being run from a collection of servers (multiple instances of the same Node running against the same database), then locking should be enable to prevent database contention. The following runtime properties control the behavior of the process.

start.synctriggers.job

Whether the sync triggers job is enabled for this node. [Default: true]

job.synctriggers.aftermidnight.minutes

If scheduled, the sync triggers job will run nightly. This is how long after midnight that job will run. [Default: 15]

cluster.lock.during.sync.triggers

Indicate if the sync triggers job is clustered and requires a lock before running. [Default: false]

7.4. Enabling and Disabling Synchronization

7.5. Java Management Extensions

Monitoring and administrative operations can be performed using Java Management Extensions (JMX). SymmetricDS uses MX4J to expose JMX attributes and operations that can be accessed from the built-in web console, Java's jconsole, or an application server. By default, the web management console can be opened from the following address:

```
http://localhost:31416/
```

Using the Java `jconsole` command, SymmetricDS is listed as a local process named `SymmetricLauncher`. In `jconsole`, SymmetricDS appears under the MBeans tab as `DefaultDomain`.

The management interfaces under `DefaultDomain` are organized as follows:

- Node - administrative operations
- Incoming - statistics about incoming batches
- Outgoing - statistics about outgoing batches
- Parameters - access to properties set through the parameter service
- Notifications - setting thresholds and receiving notifications

7.6. Opening Registration

7.7. Viewing Batches

7.8. Statistics

7.9. Clustering

A single SymmetricDS node may be clustered across a series of instances, creating a web farm. A node might be clustered to provide load balancing and failover, for example.

When clustered, a hardware load balancer is typically used to round robin client requests to the cluster. The load balancer should be configured for stateless connections. Also, the `sync.url` SymmetricDS property should be set to the URL of the load balancer.

If the cluster will be running any of the SymmetricDS jobs, then the following properties should be set to `true`.

```
cluster.lock.during.purge=true
cluster.lock.during.pull=true
cluster.lock.during.push=true
cluster.lock.during.heartbeat=true
cluster.lock.during.sync.triggers=true
```

By setting these properties to `true`, SymmetricDS will use a row in the `SYM_LOCK` table as a semaphore to make sure that only one instance at a time runs a job. When a lock is acquired, a row is updated in the lock table with the time of the lock and the server id of the locking job. The lock time is set back to null

when the job is finished running. Another instance of SymmetricDS cannot acquire a lock until the locking instance (according to the server id) releases the lock. If an instance is terminated while the lock is still held, an instance with the same server id is allowed to re-acquire the lock. If the locking instance remains down, the lock can be broken after a period of time, specified by the `cluster.lock.timeout.ms` property, has expired. Note that if the job is still running and the lock expires, two jobs could be running at the same time which could cause database deadlocks.

By default, the locking server id is the hostname of the server. If two clustered instances are running on the same server, then the `cluster.server.id` property may be set to indicate the name that the instance should use for its server id.

When deploying SymmetricDS to an application server like Tomcat or JBoss, no special session clustering needs to be configured for the application server.

7.10. Running as a Windows Service

SymmetricDS uses a Java service wrapper called [JavaService](#) in order to run in the background as a Windows NT/2000/XP system service. The JavaService executable is named `sym_service.exe` so it can be easily identified from a list of running processes. To install the service, use the provided script:

```
bin\install_service.bat
```

The script contains configuration that can be changed, including the server port number and file locations. The server defaults to port 8080. It will look in the `conf` directory for the `symmetric.properties` file and the `log4j.xml` file. Logging for standard out, error, and application are written to the `logs` directory.

When the configuration is changed, the service should be un-installed and then installed again. To un-install the service, use the provided script:

```
bin\uninstall_service.bat
```

Use the **net** command to start and stop the service:

```
net start symmetric
```

```
net stop symmetric
```

7.11. Running as a Unix Service

SymmetricDS uses an init script in order to run in the background as a Unix service (also called a daemon). The `symmetric.init` file follows the Linux Standard Base specification, which should work on many systems, including Fedora and Debian. To install the script, copy it into the system init directory:

```
cp bin/symmetric.init /etc/init.d/symmetric
```

Edit the init script to set the `SYM_HOME` variable to the directory where SymmetricDS is located. The init script calls the `sym_service` script, which contains configuration such as port number and file locations. The server defaults to port 8080. It will look in the `conf` directory for the `symmetric.properties` file and the `log4j.xml` file. Logging for standard out, error, and application are written to the `logs` directory.

To enable the service to run automatically when the system is started:

```
/sbin/chkconfig --add symmetric
```

To disable the service from running automatically:

```
/sbin/chkconfig --del symmetric
```

Use the **service** command to start, stop, and query the status of the service:

```
/sbin/service symmetric start
```

```
/sbin/service symmetric stop
```

```
/sbin/service symmetric status
```

Appendix A. Parameters

There are two kinds of parameters that can be used to configure the behavior of SymmetricDS: *Startup Parameters* and *Runtime Parameters*. Startup Parameters are required to be in a system property or a property file, while Runtime Parameters can also be found in the Parameter table from the database. Parameters are re-queried from their source at a configured interval and can also be refreshed on demand by using the JMX API. The following table shows the source of parameters and the hierarchy of precedence.

Table A.1. Parameter Locations

Location	Required	Description
<i>symmetric-default.properties</i>	Y	Packaged inside symmetric-ds.jar file. This file has all the default settings along with descriptions.
<i>symmetric.properties</i>	N	Provided by the end user on the classpath. The first symmetric.properties found on the classpath will be used.
<i>symmetric.properties</i>	N	Provided by the end user in the current system user's user.home directory.
<i>named properties file 1</i>	N	Provided by the end user as a Java system property (i.e. -Dsymmetric.override.properties.file.1=file://my.properties) or in the constructor of a <code>SymmetricEngine</code> .
<i>named properties file 2</i>	N	Provided by the end user as a Java system property (i.e. -Dsymmetric.override.properties.file.2=classpath://my.properties) or in the constructor of a <code>SymmetricEngine</code> .
<i>Java System Properties</i>	N	Any SymmetricDS property can be passed in as a -D property to the runtime. It will take precedence over any properties file property.
<i>Parameter table</i>	N	A table which contains SymmetricDS parameters. Parameters can be targeted at a specific node group and even at a specific external id. These settings will take precedence over all of the above.
<i>IParameterFilter</i>	N	An extension point which allows parameters to be sourced from another location or customized. These settings will take precedence over all of the above.

A.1. Startup Parameters

Startup parameters are read once from properties files and apply only during start up. The following properties are used:

db.jndi.name

The name of the database connection pool to use, which is registered in the JNDI directory tree of the application server. It is recommended that this DataSource is NOT transactional, because SymmetricDS will handle its own transactions. If NOT using a JNDI connection pool, you must provide information about the database connection using the `db.driver`, `db.url`, `db.user`, and `db.password` properties instead, which will create a pool of connections using the `db.pool` properties.

[Default:]

db.driver

The class name of the JDBC driver. If `db.jndi.name` is set, this property is ignored.

[Default: `com.mysql.jdbc.Driver`]

db.url

The JDBC URL used to connect to the database. If `db.jndi.name` is set, this property is ignored.

[Default: `jdbc:mysql://localhost/symmetric`]

db.user

The database username, which is used to login, create, and update SymmetricDS tables. To use an encrypted username, see the section Encrypted Passwords. If `db.jndi.name` is set, this property is ignored. [Default: `symmetric`]

db.password

The password for the database user. To use an encrypted password, see the section Encrypted Passwords. If `db.jndi.name` is set, this property is ignored. [Default:]

db.pool.initial.size

The initial size of the connection pool. If `db.jndi.name` is set, this property is ignored. [Default: 5]

db.pool.max.active

The maximum number of connections that will be allocated in the pool. If `db.jndi.name` is set, this property is ignored. [Default: 10]

db.pool.max.wait.millis

This is how long a request for a connection from the datasource will wait before giving up. If `db.jndi.name` is set, this property is ignored. [Default: 30000]

db.pool.min.evictable.idle.millis

This is how long a connection can be idle before it will be evicted. If `db.jndi.name` is set, this property is ignored. [Default: 120000]

db.spring.bean.name

The name of a Spring bean to use as the DataSource. If you want to use a different DataSource other than the provided DBCP version that SymmetricDS uses out of the box, you may set this to be the Spring bean name of your DataSource.

db.sql.query.timeout.seconds

The timeout in seconds for queries running on the database. [Default: 60]

db.tx.timeout.seconds

This is how long the default transaction time is. This needs to be fairly big to account for large data loads. [Default: 7200]

db.jdbc.streaming.results.fetch.size

This is the default fetch size for streaming result sets into memory from the database.

[Default: 1000]

auto.config.database

If this is true, the configuration and runtime tables used by SymmetricDS are automatically created during startup. [Default: true]

auto.upgrade

If this is true, when symmetric starts up it will try to upgrade tables to latest version. [Default: true]

auto.sync.configuration

If this is true, create triggers for the SymmetricDS configuration table that will synchronize changes to node groups that pull from the node where this property is set. [Default: true]

sync.table.prefix

When symmetric tables are created and accessed, this is the prefix to use for the table name.
[Default: sym]

trigger.prefix

Set this if the trigger names need to be prefixed. This is useful when running two symmetric instances against the same database. [Default:]

engine.name

This is the engine name. This should be set if you have more than one engine running in the same JVM. It is used to name the JMX management bean. [Default: Default]

start.push.job

Whether the push job is enabled for this node. [Default: true]

start.pull.job

Whether the pull job is enabled for this node. [Default: true]

start.purge.job

Whether the purge job is enabled for this node. [Default: true]

start.synctriggers.job

Whether the sync triggers job is enabled for this node. [Default: true]

start.heartbeat.job

Whether the heartbeat job is enabled for this node. The heartbeat job simply inserts an event to update the heartbeat_time column on the node table for the current node. [Default: true]

job.purge.period.time.ms

This is how often the purge job will be run. [Default: 600000]

job.statflush.period.time.ms

This is how often accumulated statistics will be flushed out to the database from memory.
[Default: 600000]

A.2. Runtime Parameters

Runtime parameters are read periodically from properties files or the database. The following properties are used:

auto.registration

If this is true, registration is opened automatically for nodes requesting it. [Default: false]

auto.reload

If this is true, a reload is automatically sent to nodes when they register. [Default: false]

http.download.rate.kb

This is the download rate for the HTTP symmetric transport. A value of -1 means full throttle. [Default: -1]

http.concurrent.workers.max

This is the number of HTTP concurrent push/pull requests symmetric will accept. This is controlled by the NodeConcurrencyFilter. [Default: 20]

outgoing.batches.peek.ahead.window.after.max.size

This is the maximum number of events that will be peeked at to look for additional transaction rows after the max batch size is reached. The more concurrency in your db and the longer the transaction takes the bigger this value might have to be. [Default: 100]

incoming.batches.skip.duplicates

Whether or not to skip duplicate batches that are received. A duplicate batch is identified by the batch ID already existing in the incoming batch table. If this happens, it means an acknowledgement was lost due to failure or there is a bug. Accepting a duplicate batch in this case can mean overwriting data with old data. Another cause of duplicates is when the batch sequence number is reset, which might happen in a lab environment. Skipping a duplicate batch in this case would prevent data changes from loading. Generally, in a production environment, this setting should be true. [Default: true]

num.of.ack.retries

This is the number of times we will attempt to send an ACK back to the remote node when pulling and loading data. [Default: 5]

time.between.ack.retries.ms

This is the amount of time to wait between trying to send an ACK back to the remote node when pulling and loading data. [Default: 5000]

dataloader.lookup.target.schema

Use the target schema specified on sym_trigger when loading data. [Default: false]

dataloader.enable.fallback.update

If an insert is received, but the row already exists, then try an update instead. [Default: true]

dataloader.enable.fallback.insert

If an update is received, but it affects no rows, then try to insert instead. [Default: true]

dataloader.allow.missing.delete

If a delete is received, but it affects no rows, then continue. [Default: true]

cluster.server.id

Set this if you want to give your server a unique name to be used to identify which server did what action. Typically useful when running in a clustered environment. This is currently used by the

ClusterService when locking for a node. [Default:]

cluster.lock.timeout.ms

Time limit of lock before it is considered abandoned and can be broken. [Default: 1800000]

cluster.lock.during.purge

Indicate if the purge job is clustered and requires a lock before running. [Default: false]

cluster.lock.during.pull

Indicate if the pull job is clustered and requires a lock before running. [Default: false]

cluster.lock.during.push

Indicate if the push job is clustered and requires a lock before running. [Default: false]

cluster.lock.during.heartbeat

Indicate if the heartbeat job is clustered and requires a lock before running. [Default: false]

cluster.lock.during.sync.triggers

Indicate if the sync triggers job is clustered and requires a lock before running. [Default: false]

initial.load.delete.first

Set this if tables should be purged prior to an initial load. [Default: false]

initial.load.create.first

Set this if tables (and their indexes) should be created prior to an initial load. [Default: false]

http.timeout.ms

Sets both the connection and read timeout on the internal HttpURLConnection. [Default: 600000s]

http.compression

Whether or not to use compression over HTTP connections. Currently, this setting only affects the push connection of the source node. Compression on a pull is enabled using a filter in the web.xml for the PullServlet. [Default: true]

job.random.max.start.time.ms

When starting jobs, symmetric attempts to randomize the start time to spread out load. This is the maximum wait period before starting a job. [Default: 10000]

purge.retention.minutes

This is the retention for how long synchronization data will be kept in the SymmetricDS synchronization tables. Note that data will be purged only if the purge job is enabled. [Default: 7200]

job.push.period.time.ms

This is how often the push job will be run. [Default: 60000]

job.pull.period.time.ms

This is how often the pull job will be run. [Default: 60000]

job.synctriggers.aftermidnight.minutes

If scheduled, the sync triggers job will run nightly. This is how long after midnight that job will run. [Default: 15]

schema.version

This is hook to give the user a mechanism to indicate the schema version that is being synchronized. This property is only valid if you use the default IRuntimeConfiguration implementation. [Default: ?]

registration.url

The URL where this Node can connect for registration to receive its configuration. This property is only valid if you use the default IRuntimeConfiguration implementation. [Default:]

my.url

The URL where this Node can be contacting for synchronization. This property is only valid if you use the default IRuntimeConfiguration implementation. [Default: http://localhost:8080/sync]

group.id

The Node Group ID for this Node. This property is only valid if you use the default IRuntimeConfiguration implementation. [Default: default]

external.id

The secondary identifier for this Node that has meaning to the system where it is deployed. While the Node ID is a generated sequence number, the external ID could have meaning in the user's domain, such as a retail store number. This property is only valid if you use the default IRuntimeConfiguration implementation. [Default:]

transport.type

Specify the transport type. Supported values currently include: http, internal. [Default: http]

hsqldb.initialize.db

If using the HsqlDbDialect, this property indicates whether Symmetric should setup the embedded database properties or if an external application will be doing so. [Default: true]

Appendix B. Database Notes

Each database management system has its own characteristics that results in feature coverage in SymmetricDS. The following table shows which features are available by database.

Table B.1. Support by Database

Database	Versions supported	Transaction Identifier	Fallback Update	Conditional Sync	Update Loop Prevention	BLOB Sync
Oracle	8.1.7 and above	Y	Y	Y	Y	Y
MySQL	5.0.2 and above	Y	Y	Y	Y	Y
PostgreSQL	8.2.5 and above	Y (8.3 and above only)	Y	Y	Y	Y
SQL Server	2005	Y	Y	Y	Y	Y
HSQLDB	1.8	Y	Y	Y	Y	Y
H2	1.1	Y	Y	Y	Y	Y
Apache Derby	10.3.2.1	Y	Y	Y	Y	Y
IBM DB2	9.5	N	Y	Y	Y	Y
Firebird	2.0	Y	Y	Y	Y	Y

B.1. Oracle

On Oracle Real Application Clusters (RAC), sequences should be ordered so data is processed in the correct order. To offset the performance cost of ordering, the sequences should also be cached.

```
alter sequence SEQ_SYM_DATA_DATA_ID cache 1000 order;  
alter sequence SEQ_SYM_OUTGOIN_BATCH_BATCH_ID cache 1000 order;  
alter sequence SEQ_SYM_TRIGGER_RIGGER_HIST_ID cache 1000 order;  
alter sequence SEQ_SYM_TRIGGER_TRIGGER_ID cache 1000 order;
```

While BLOBs are supported on Oracle, the LONG data type is not. LONG columns cannot be accessed from triggers.

The SymmetricDS user generally needs privileges for connecting and creating tables (including indexes), triggers, sequences, and procedures (including packages and functions). The following is an example of the needed grant statements:

```
GRANT CONNECT TO SYMMETRIC;  
GRANT RESOURCE TO SYMMETRIC;  
GRANT CREATE ANY TRIGGER TO SYMMETRIC;
```

```
GRANT EXECUTE ON UTL_RAW TO SYMMETRIC;
```

B.2. MySQL

MySQL supports several storage engines for different table types. SymmetricDS requires a storage engine that handles transaction-safe tables. The recommended storage engine is InnoDB, which is included by default in MySQL 5.0 distributions. Either select the InnoDB engine during installation or modify your server configuration. To make InnoDB the default storage engine, modify your MySQL server configuration file (`my.ini` on Windows, `my.cnf` on Unix):

```
default-storage_engine = innodb
```

Alternatively, you can convert tables to the InnoDB storage engine with the following command:

```
alter table t engine = innodb;
```

On MySQL 5.0, the SymmetricDS user needs the SUPER privilege in order to create triggers.

```
grant super on *.* to symmetric;
```

On MySQL 5.1, the SymmetricDS user needs the TRIGGER and CREATE ROUTINE privileges in order to create triggers and functions.

```
grant trigger on *.* to symmetric;
```

```
grant create routine on *.* to symmetric;
```

B.3. PostgreSQL

Starting with PostgreSQL 8.3, SymmetricDS supports the transaction identifier.

In order to function properly, SymmetricDS needs to use session variables. On PostgreSQL, session variables are enabled using a custom variable class. Add the following line to the `postgresql.conf` file of PostgreSQL server:

```
custom_variable_classes = 'symmetric'
```

This setting is required, and SymmetricDS will log an error and exit if it is not present.

Before database triggers can be created by in PostgreSQL, the plpgsql language handler must be installed on the database. The following statements should be run by the administrator on the database:

```
CREATE FUNCTION plpgsql_call_handler() RETURNS language_handler AS
    '$libdir/plpgsql' LANGUAGE C;

CREATE FUNCTION plpgsql_validator(oid) RETURNS void AS
    '$libdir/plpgsql' LANGUAGE C;

CREATE TRUSTED PROCEDURAL LANGUAGE plpgsql
    HANDLER plpgsql_call_handler
    VALIDATOR plpgsql_validator;
```

B.4. MS SQL Server

SQL Server was tested using the [jTDS](#) JDBC driver.

B.5. HSQLDB

HSQLDB was implemented with the intention that the database be run embedded in the same JVM process as SymmetricDS. Instead of dynamically generating static SQL-based triggers like the other databases, HSQLDB triggers are Java classes that re-use existing SymmetricDS services to read the configuration and insert data events accordingly.

The transaction identifier support is based on SQL events that happen in a 'window' of time. The trigger(s) track when the last trigger fired. If a trigger fired within X milliseconds of the previous firing, then the current event gets the same transaction identifier as the last. If the time window has passed, then a new transaction identifier is generated.

B.6. H2

The H2 database allows only Java-based triggers. Therefore the H2 dialect requires that the SymmetricDS jar file be in the database's classpath.

B.7. Apache Derby

The Derby database can be run as an embedded database that is accessed by an application or a standalone server that can be accessed from the network. This dialect implementation creates database triggers that make method calls into Java classes. This means that the supporting JAR files need to be in the classpath when running Derby as a standalone database, which includes `symmetric-ds.jar` and `commons-lang.jar`.

B.8. IBM DB2

The DB2 Dialect uses global variables to enable and disable node and trigger synchronization. These variables are created automatically during the first startup. The DB2 JDBC driver should be placed in the "lib" folder.

Currently, the DB2 Dialect for SymmetricDS does not provide support for transactional synchronization. Large objects (LOB) are supported, but are limited to 16,336 bytes in size. The current features in the DB2 Dialect have been tested using DB2 9.5 on Linux and Windows operating systems.

B.9. Firebird

The Firebird Dialect requires the installation of a User Defined Function (UDF) library in order to provide functionality needed by the database triggers. SymmetricDS includes the required UDF library, called SYM_UDF, in both source form (as a C program) and as pre-compiled libraries for both Windows and Linux. The SYM_UDF library is copied into the UDF folder within the Firebird installation directory.

For Linux users:

```
cp databases/firebird/sym_udf.so /opt/firebird/UDF
```

For Windows users:

```
copy databases\firebird\sym_udf.dll C:\Program Files\Firebird\Firebird_2_0\UDF
```

The [Jaybird JDBC driver](#) was used during testing, but the user must download the driver and place it in the SymmetricDS "lib" folder.

The following limitations currently exist for this dialect:

- The outgoing batch does not honor the channel size, and all outstanding data events are included in a batch.
- Syncing of Binary Large Object (BLOB) is limited to 16K bytes per column.
- Syncing of character data is limited to 32K bytes per column.

Appendix C. Data Format

The SymmetricDS Data Format is used to stream data from one node to another. The data format reader and writer are pluggable with an initial implementation using a format based on Comma Separated Values (CSV). Each line in the stream is a record with fields separated by commas. String fields are surrounded with double quotes. Double quotes and backslashes used in a string field are escaped with a backslash. Binary values are represented as a string with hex values in "\0xab" format. The absence of any value in the field indicates a null value. Extra spacing is ignored and lines starting with a hash are ignored.

The first field of each line gives the directive for the line. The following directives are used:

version, {major},{minor},{patch}

Indicates the version of the file format

table, {table name}

Sets the context of which table on which to operate.

keys, {column name...}

Lists the column names that are used as the primary key for the table. Only needs to occur after the first occurrence of the table.

columns, {column name...}

Lists all the column names (including key columns) of the table. Only needs to occur after the first occurrence of the table.

insert, {column value...}

Insert into the table with the values that correspond with the columns.

update, {new column value...},{old key value...}

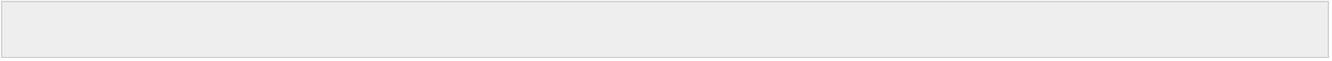
Update the table using the old key values to set the new column values.

delete, {old key value...}

Delete from the table using the old key values.

Example C.1. Data Format Stream

```
version, 1,0,0
table, item_selling_price
keys, price_id
columns, price_id, price, cost
insert, 55, 0.65, 0.55
table, item
keys, item_id
columns, item_id, price_id, name
insert, 110000055, 55, "Soft Drink"
delete, 110000001
table, item_selling_price
update, 55, 0.75, 0.65, 55
```



Appendix D. Version Numbering

The software is released with a version number based on the [Apache Portable Runtime Project](#) version guidelines. In summary, the version is denoted as three integers in the format of MAJOR.MINOR.PATCH. Major versions are incompatible at the API level, and they can include any kind of change. Minor versions are compatible with older versions at the API and binary level, and they can introduce new functions or remove old ones. Patch versions are perfectly compatible, and they are released to fix defects.